JOHANNES KEPLER
UNIVERSITY LINZ

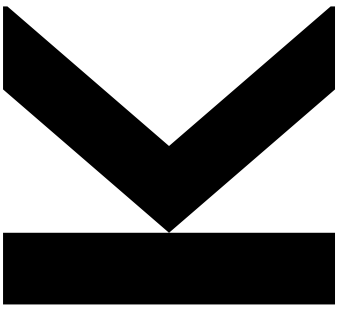
Author
Hartl Konstantin

Submission
Institute of Networks and Security

Thesis Supervisor
Assoc.Prof.
Mag. Dipl.-Ing.
Dr. Sonntag Michael

Linz, February 2019

A program analyzing the fragmentation of files



Bachelor's Thesis
to confer the academic degree of
Bachelor of Science
in the Bachelor's Program
Informatics

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

File fragmentation is generally known because of its performance-degrading nature on HDDs. For file recovery or digital forensics it can be difficult to find and restore all fragments, if not impossible. This paper provides insight into an implementation of a program gathering information about the fragmentation of files. The goal is to save and transmit the findings to a central repository to be able to generate statistics over different operating systems, file systems and specific file types. The paper starts off with theoretical concepts of fragments and then dives into the functions different operating system provide and how they are used. Direct results about fragmentation of actual file systems are outside the scope of this thesis.

Dateifragmentierung ist generell aufgrund von Performanceeinbußen auf HDDs bekannt. Für Dateiwiederherstellung und digitale Forensik ist es aber schwer bzw. unmöglich, Informationen über alle Fragmente einer Datei zu erfassen. Dieses Paper gibt Einsicht in die Implementierung eines Programms, das Informationen über die Fragmentierung von Dateien sammelt. Das Ziel ist, eine zentrale Sammlung zu erstellen, um daraus Statistiken über verschiedene Betriebssysteme, Dateisysteme und spezielle Dateitypen zu erstellen. Das Paper beginnt mit Theorie über Dateifragmente und geht im Folgenden auf die Funktionen verschiedener Betriebssysteme ein, die für diesen Zweck bereitgestellt werden. Ergebnisse über Dateifragmentierung sind außerhalb des Rahmens dieses Papers.

Contents

1
Introduction

File-fragmentation occurs because the file system has to deal with file deletions and unknown-length writes on a daily basis. Meanwhile, only limited storage space is available but it also should be able to be used to the fullest extent.

Deleting a file or reducing its size leaves an unallocated hole behind [1][2]. This gap can only be used to store a file of the same size or smaller. Over time the number and size of these holes could increase dramatically [1][2]. In the worst case this can escalate to a level where a bigger file cannot be stored although the sum of the sizes of the holes exceeds the size of the file to be stored.

To solve this problem, fragmentation is introduced which allows the file to be broken up into parts [1][2] which in turn are written in the holes. Reading the complete file requires visiting each of the locations where the fragments are stored and reassembling them to the complete file [3].

This is where the performance problems occur on HDDs. The read-write-head needs to jump to each of the positions in order to read or write the fragments [1][2]. Without fragments, the head would only have to jump to the beginning of the file and read/write continuously until the end of the file without any additional jumps.

SSDs do not have the problem with access time [1]. Their problem is that the physical storage can only sustain a limited number of writes which is counteracted by wear-leveling [1]. It introduces deliberate fragmentation to distribute the writes evenly between physical storage units [1].

Fragments not only decrease the read-write-performance but also hinder file recovery [1] or digital forensics [3]. A successful recovery process must yield the bytes in the same order (and possibly also the same length, but this depends on the file format). Recovering files is a difficult process [1] and there are multiple ways to tackle this problem. We will not get into details as this is outside of the scope of this paper.

A sophisticated approach is to search for files without a file-table and only by the data itself [2]. Taking the file-table into account, this can be restricted to the

"unallocated" space that also contains deleted files [2]. The search-space can be extended to the whole storage medium, ignoring a file-table that might or might not exist [2]. This process is called *file carving* [2].

Each storage block is scanned for headers (and possibly footers if applicable) [2]. However, this is only possible for known and supported file types by the recovering application. There is also the possibility to look for a file magic which is a well-known byte-sequence that can be mapped to a specific file type. This method only finds the first block as the following blocks on disk might be from different files. Other fragments must be determined by the file structure [1].

1.1 Terminology

Before continuing on with the technical background, we would like to introduce some important terminology.

Sector, block, cluster

Sector, block and cluster are fundamental terms and concepts on how files are stored on a storage medium. Having spent a lot of time researching in different online forums about the topic of file systems, these words are often used interchangeably.

A *sector* is the smallest addressable unit of information on a storage device. It can hold any information and can only be read or written to as a whole. For current HDDs (hard disk drive) the size was typically 512 [4]. Modern drive use physical 4096 byte sector sizes [5]. However, a HDD with a larger physical sector size may appear to have the typical 512 byte sectors via translation [4][5].

A *block* generally just describes a collection of bytes. In the Linux file system world, a block refers to the amount of data a file system can handle for storage of data. This is done to reduce the amount of necessary overhead required to organize files [6]. The block size is a multiple of the (original) 512 byte sector size to be able to read or write the exact amount of data.

By default *ext4* uses 4 KiB blocks but it is possible to specify the usage from 1 KiB up to 64 KiB [6] depending on the use-case.

The term *cluster* is mostly used in the windows world, having the same meaning as a *block*. It is the minimal amount of storage space that can be allocated by the file system to hold (a part of a) file [7].

Sparse file

A file is called sparse if it has "holes" in the allocation on disk [8]. Then the virtual size of the file is larger than the one on disk, saving space and access time.

As an example, in NTFS clusters (blocks) can be set to sparse in case they only contain zeroes and need not actually be allocated (or unallocated if previously allocated). Read operations on sparse regions simply return zeroes and are faster because no actual read operation is required from disk. However, this feature must be directly used by the calling application and is not automatically used by the file system [9].

2
Technical background

In this chapter we are going into the details on how to gather information about the fragmentation of a file. We introduce the most important functions of Windows and Linux provide and how they are used. Then we continue on how to assemble the fragments.

2.1 Gathering information about file fragments

In general, the available space on a disk is divided into partitions. These in turn contain a file system which is responsible for storing and retrieving files. In our case we are not necessarily interested where exactly the files lie on disk. We are interested in the fragmentation which we also can determine with an abstraction layer (like the clusters on Windows). This still carries the positional data of fragments in reference to some point on the disk while not knowing where this reference point exactly lies[1].

2.1.1 Windows

On Windows the distribution of files' parts are exposed via Clusters and Extents [10]. It is a high-level interface that is utilized for all file systems supported by Windows.

Clusters convey the information of the positional data of the parts of a file on disk but also internally [10]. Extents are a group of clusters directly adjacent to each other [10].

[1]For example, the reference point might be directly after the file table, at the start or the partition or even the at the beginning of the disk. But this does not matter for our purposes.

Clusters

A cluster is the minimal allocation unit for storage space in the file system [7]. The data of a file is stored in one or more clusters, although there are some exceptions. The default size for NTFS since Windows 7 is 4096 Bytes (for volumes under 16 TB) [7]. The same cluster has a different number based on the context in which it is looked at [10].

The first context is based on the offset in the file itself and is called *virtual cluster number* (VCN) [10]. Reading a file from the beginning starts from the byte-offset 0 which is also where the 0th cluster is located [10]. After reading the amount of bytes a cluster holds, the next cluster, the 1st, starts. This continues until the end of file is reached. The cluster number increments by one without any jumps which might not represent the physical storage-layout on the disk, hence virtual.

The other context is based on the position of the cluster on the volume from some reference point and is called *logical cluster number* (LCN) [10]. The volume is the host of a file system and consists of at least one partition [11].

Extents

An extent is a group of clusters that have an adjacent cluster number and are contiguous on disk [10]. An extent here is the same as the one used in Fiemap in Linux (covered in section 2.1.3). Essentially, it is a performance measure to avoid having to enumerate every cluster individually.

An example would be clusters 3, 4 and 5 that are at LCN 56, 57 and 58. All three form an extent because of their adjacent cluster numbers. The following cluster 6 at LCN 456 cannot be part of this extent because it is not contiguous on disk. Therefore, it is contained in another extent. Also, cluster 7 at 59 cannot be a part of the extent because the cluster number is not adjacent to 3, 4 or 5.

Since we are really interested in fragments it is easy to assume that extents and fragments are the same because of their similarities. Although an extent groups contiguous clusters together, there is no guarantee that an extent captures a full fragment. Especially for large files it is common that a single fragment consists of multiple extents which, of course, are contiguous and so could theoretically be combined to a single extent. We will go into details on what fragments are and how to count them in section 2.3.

An extent can also hold the special LCN value of -1 [12]. A program has the ability to reserve space for a file without having to write anything into it. This can result in a fully or partially unallocated space which is known as a sparse file. The extents without any data then return -1 indicating that no data has been written

(yet) [12] which is also known as virtual extent/cluster. Another possibility for -1 is the usage of a compression feature of the file system [12].

2.1.2 Important functions in Windows

In this section we introduce the most important functions that are used to gather the fragmentation information of a file. We go into detail which values are passed to them for our purposes and how they are depend on each other.

CreateFile

The most basic function for our need is *CreateFile* which acquires a file handle for a specific file [13]. We use this to open the file for reading. This is not restricted to the content of the file but also allows to access the metadata [13]. In our case we are only interested in the latter but to determine the file type the first bytes could be read too.

Beyond the path to the file the function requires multiple other parameters. Most of which are not important for our purposes and are only given default values. The following listing shows the C definition [13]:

```
HANDLE CreateFileA(
    LPCSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);
```

Normally, `GENERIC_READ` and/or `GENERIC_WRITE` are used for *DesiredAccess* [13] to access the data of the file. We chose it to be `0` which only allows querying the metadata [13]. The advantage is that it works even when opening with `GENERIC_READ` would have failed [13]. Presumably, this is the case when the file was opened with exclusive access by another program.

For *ShareMode* we used `FILE_SHARE_READ` which prevents us from opening a file that is currently opened by another program for writing or deletion. This helps us because the file cannot be changed while opened making the scanning process easier but it also means that we miss out on some files.

Ideally, `FILE_SHARE_READ|FILE_SHARE_WRITE|FILE_SHARE_DELETE` would be used to access the file in any case. However, we are not sure how the extent-gathering functions are impacted by another program writing to the opened file or even deleting it.

We left the *SecurityAttributes* `NULL` because we do not inherit the handle to child processes and operate fine with the default security descriptor.

The *CreationDisposition* states how to handle (non-)existence of the requested file [13]. In our case we just want to *OPEN_EXISTING*.

For *FlagsAndAttributes* we use `FILE_ATTRIBUTE_NORMAL` because it is the recommended default [13] and we do not need any of the provided functionality.

The *TemplateFile* is irrelevant for us because we do not create files and therefore leave it `NULL`.

CloseHandle

We are expected to free the handle [14] afterwards that we got from *CreateFile*.

DeviceIoControl

DeviceIoControl is a multi-purpose function that changes its functionality base on the *IoControlCode* passed to it [15]. In our program we only use the control code `FSCTL_GET_RETRIEVAL_POINTERS` to fetch extents for a given file [16].

For this particular control code, the function parameters are given as follows [16]:

```
DeviceIoControl(
    (HANDLE) hDevice,
    FSCTL_GET_RETRIEVAL_POINTERS,
    (LPVOID) lpInBuffer,
    (DWORD) nInBufferSize,
    (LPVOID) lpOutBuffer,
    (DWORD) nOutBufferSize,
    (LPDWORD) lpBytesReturned,
    (LPOVERLAPPED) lpOverlapped
);
```

The first parameter is the handle we obtain through *CreateFile* and the second one is the fixed control code [16]. We are not interested in the last tow parameters *BytesReturned* and *Overlapped*. We use an unused variable for the first and set the second to `NULL`.

As the third (and fourth) argument a `STARTING_VCN_INPUT_BUFFER` struct is expected [16]. It has only one member, the *StartingVcn*, which sets the starting offset in the file from which extents should be returned [17]. At the start of the file this would be set to 0 but it is useful if the result structure cannot hold all extents and multiple calls have to be made. It is used to resume operation and retrieve the following extents.

The fifth (and sixth) parameter of *DeviceIoControl* is a `RETRIEVAL_POINTERS_BUFFER` struct [16]. It is filled by the function with the extent information of the file [18]. The struct definition looks as follows [18]:

```
typedef struct RETRIEVAL_POINTERS_BUFFER {
    DWORD ExtentCount;
    LARGE_INTEGER StartingVcn;
    struct {
        LARGE_INTEGER NextVcn;
        LARGE_INTEGER Lcn;
    } Extents[1];
} RETRIEVAL_POINTERS_BUFFER, *PRETRIEVAL_POINTERS_BUFFER;
```

The *StartingVcn* contains the VCN at which the first extent starts [18]. This can be different than the value set in `STARTING_VCN_INPUT_BUFFER` [18]. At normal operation these should be identical because we can calculate the next value based on the last returned extent. However, a random (but valid) VCN can be requested which will be "rounded down" to the start of an extent the given VCN is contained in [18].

The last member is an array named *Extents* which is only one element long (in contrast to fiemap on Linux!) and its type is an anonymous struct [18]. This struct holds the next VCN and the LCN of the extent [18].

The important point here is that the array is at the end of the buffer. This allows the array to be potentially larger than the length of 1, contrary to what the buffer-struct definition states. Therefore, we can allocate the size of the buffer struct plus additional storage for more array elements. The additional space is communicated to Windows by the *OutBufferSize* parameter by passing the real, full struct size in bytes [16]. The size in C can be calculated by `sizeof(RETRIEVAL_POINTERS_BUFFER) + (sizeof(LARGE_INTEGER) * 2) * (nExtents - 1)`.

The *ExtentCount* in the buffer struct states how many extents have been returned by the function call [18]. Since it is not known how many extents will be returned and the struct can be allocated once and then re-used, it must be used for iterating over the results.

GetVolumePathName

The *GetVolumePathName* retrieves the root path of the file system for a specific file [19]. This might be `C:\` for a file `C:\test.txt` on an ordinary file system [19]. If a file system is mounted on the path of another one, the output could also be `C:\mountpoint` for an example path `C:\mountpoint\testfile.text` [19].

We do not need the root path for our purposes directly but *GetVolumeInformation* and *GetDiskFreeSpace* take it as argument.

GetVolumeInformation

GetVolumeInformation exposes some meta details about a mounted file system [20]. Though, we are only interested in the *FileSystemName*.

The first argument is the *RootPathName* [20] which is the root path we get as a result from *GetVolumePathName*. The last argument(s) is the *FileSystemName* [20] which we are interested in. Since we do not need most of the remaining information, we simply set the other parameters to `NULL`.

GetDiskFreeSpace

The extent sizes are given in clusters. To query the size of a cluster, *GetDiskFreeSpace* can be used [21]. The following listing shows the C definition [21]:

```
BOOL GetDiskFreeSpaceA(
    LPCSTR   lpRootPathName,
    LPDWORD lpSectorsPerCluster,
    LPDWORD lpBytesPerSector,
    LPDWORD lpNumberOfFreeClusters,
    LPDWORD lpTotalNumberOfClusters
);
```

The first argument *RootPathName* [21] expects the result of *GetVolumePathName*. The second and third, *SectorsPerCluster* and *BytesPerSector* [21], are the ones we are interested in. By multiplying them we get the number of bytes per cluster. The remaining two are the free and total number of clusters [21] which we incorporate later in our report.

2.1.3 Linux

On Linux there are two ways on gathering the spatial information of files: Fibmap and Fiemap.

Fiemap is quite similar to the approach in Windows, where the result is a set of extents which group contiguous blocks together [22]. Fibmap is the simplistic way of retrieving every single block one by one.

Fiemap is preferred because the grouping performs better [23][22] than having to request and process each and every block of a file individually. Fiemap must be supported and implemented by the file system driver explicitly [22]. Since not all of the numerous file systems that exist for Linux have Fiemap implemented, the fallback to the slow Fibmap is the only option available. The tool named *filefrag* does this by default [23].

Fiemap

Fiemap is similar to Windows in terms of the result structs and the function used. On Linux there is also a general-purpose function `ioctl` that changes its function based on the second parameter [22][24]. In our use-case the function looks like this [24]:

```
int ioctl(int fd, unsigned long int request, struct fiemap *fiemap)
```

The first argument is the file descriptor we get from the `open` function [24]. The second parameter is the request code which in our case is always `FS_IOC_FIEMAP` and the third and final is the pointer to the fiemap buffer struct [24]. The function result is negative in case of an error.

The fiemap struct holds both input and output values which is different from Windows where two separate structs were used. The fiemap struct has the following layout [22]:

```
struct fiemap {
    __u64    fm_start;
    __u64    fm_length;
    __u32    fm_flags;
    __u32    fm_mapped_extents;
    __u32    fm_extent_count;
    __u32    fm_reserved;
    struct fiemap_extent fm_extents[0];
};
```

fm_start signifies the start from which file offset (in bytes) the extents should be returned [22]. For the first call we set it to `0` and then calculate the value for the next iteration based on the last extent returned.

fm_length specifies the length of the file (in bytes) to examine [22] starting from *fm_start*. Since we want to get as much information as possible with one call, we set it to the maximum possible value which in C can be described as `~0`.

The handling of non-exact values for *fm_start* and *fm_length* may be rounded (like on Windows). *fm_start* might be rounded down to the start of the extent the specified position falls into [22]. Meanwhile *fm_length* may be rounded up to include the full extent that would otherwise be truncated because of the length restriction give by the caller [22].

fm_flags is a special parameter because it functions as input and output simultaneously [22]. The flags specified by the caller are evaluated by the kernel and if unsupported flags are set, *EBADR* is returned and only the unsupported flags remain in this member [22]. This member does not change otherwise [22]. In our case we do not use any of the flags so we set this to `0`.

fm_mapped_extents holds the number of extents that were returned [22]. While this is the number of populated entries in the *fm_extents* array, *fm_extent_count* must be set to the length of the array [22].

fm_ extents is an array that holds the *fiemap_ extents* returned [22]. The array is defined as zero-length but since it is at the end it can extend into the memory area beyond the struct boundary. Therefore, we allocate an arbitrary amount of memory based on the number of extents we want to be able to fetch with one function call. As an example this would be the size of the base struct plus n times the size of an array element. The important difference to Windows is that this array has no elements by default while the Windows version has at least one. This has to be kept in mind when working on both implementations to not produce a off-by-one error, especially if a specific array size is wanted.

The elements in the *fm_ extents* array are given by the following structure [22]:

```
struct fiemap_extent {
    __u64   fe_logical;
    __u64   fe_physical;
    __u64   fe_length;
    __u64   fe_reserved64[2];
    __u32   fe_flags;
    __u32   fe_reserved[3];
};
```

fe_ logical holds the start of the extent as file offset (in bytes) [22].
fe_ physical holds the start of the extent as byte-offset from the beginning of the storage medium [22]. *fe_ length* states the length of the extent in bytes [22].

For *fe_ flags* there is a large list of flags that can be returned [22]. Out of all possible values, `FIEMAP_EXTENT_LAST` is the most interesting one for us because it marks the end of the file [22]. There are multiple other flags that could be checked [22]. Some of them show extended status about the allocation [22], but because of lack of documentation and no (easy) way to test we decided to ignore them.

There are two ways different ways to find out if all extents have been retrieved. The first way would be to check for the `FIELMAP_EXTENT_LAST` flag and terminate if found. Alternatively, *fm_ extent_ count* can be set to zero and *fm_ mapped_ extents* will be set to the number of extents the file consists of [22].

Fibmap

In contrast to *Fiemap*, *Fibmap* only returns as single block position per call [24][25]. The blocks have to be grouped to fragments manually.

The most important function again is *ioctl* which has the following signature in this case [24]:

```
int ioctl(int fd, unsigned long int request, int *block)
```

ioctl takes the file descriptor as first parameter and then `FIBMAP` as *request*. The third parameter is a pointer which acts both as input and output. The input is the

index of the block to query and ranges from 0 to the number of blocks the file has allocated (minus one). The result is negative in case of an error [24].

Since we are dealing with blocks, we need to know how many bytes a single one has. To query the block size, *ioctl* can be called with `FIGETBSZ`[2] as *request*. In this case the third parameter is a pointer to an int in which the result is stored.

To get the number of blocks the file consists of, *fstat* has to be called. It takes the file descriptor as first parameter and a pointer to a stat struct as second. From this struct we only extract *st_size* which is the file size in bytes. We can now calculate the number of blocks with $(st.st\_size + blocksize - 1)/blocksize$ [24].

It is now possible to iterate over all block indices and fetch the physical block positions [24]. Grouping them to extents or directly fragments has to be done completely manually.

The big downside of this approach, aside from the performance issue, is that it requires root privileges. The application implemented for this thesis therefore requires the elevated privileges if the fallback to *Fibmap* is triggered too.

Stat

With *stat* it is possible to to fetch meta information about a file. The similar function *fstat* differs in the file parameter such that it takes a file descriptor instead of a path name. The second parameter is expected to be a pointer to the following struct which is filled [26]:

```
struct stat {
    dev_t     st_dev;
    ino_t     st_ino;
    mode_t    st_mode;
    nlink_t   st_nlink;
    uid_t     st_uid;
    gid_t     st_gid;
    dev_t     st_rdev;
    off_t     st_size;
    blksize_t st_blksize;
    blkcnt_t  st_blocks;
}
```

Although there is a lot of information returned in the struct, we are mainly intested in *st_size* which holds the size of the file in bytes. It is also possible to get the block size from *st_blksize* [26].

[2]Fibmap example code generally uses ioctl with FIGETBSZ. All methods retrieving the block size should work but we are unsure about the differences. In the source of the *filefrag* utility they even overwrite *st_blksize* of the stat struct with the result of `FIGETBSZ` [24]

StatVfs

StatVfs allows to retrieve meta information about a file system. Also *fstatvfs* is available which takes a file descriptor instead of a path as the first argument. The second parameter is a pointer to the following struct [27]:

```
struct statvfs {
    unsigned long  f_bsize;
    unsigned long  f_frsize;
    fsblkcnt_t     f_blocks;
    fsblkcnt_t     f_bfree;
    fsblkcnt_t     f_bavail;
    fsfilcnt_t     f_files;
    fsfilcnt_t     f_ffree;
    fsfilcnt_t     f_favail;
    unsigned long  f_fsid;
    unsigned long  f_flag;
    unsigned long  f_namemax;
};
```

For us, the most interesting members are *f_bfree* which holds the number of free blocks, *f_blocks* which is the total size of the file system in blocks and finally the *f_frsize*[3] which is the block size [27] so.

2.2 File system links

Besides storing simple files in folders, file systems often have the ability to create special files (or folders) that link to other files (or folders). Both Linux and windows understand hard links and symbolic links but the actual availability depends on the support of the file system in use.

Such special links are not related to file fragments but they do have to be considered for our purposes. For example, it is possible to link to the same file twice which would be counted twice in our statistic. Even worse would be links to folders where all files would be counted twice. Additionally, links could be crafted in such a way that our scanner could be caught up in cyclic references.

2.2.1 Hard link

Hard links can only refer to files, not directories [28]. It is a way to create multiple paths that refer to the same file [28]. Since a hard link refers to the same file, reading and writing to one hard link is the same as writing to the same file with the same path [28]. Also the file attributes are shared [28]. Hard links are only available on the same volume [28].

[3] Based on unreliable sources, *f_bsize* seems to be the preferred block size and *f_frsize* being the "minimal" block size. The documentation explicitly states that *f_blocks* is given in *f_frsize* [27].

Since such a link has to be created manually and is only valid for files we chose to ignore this possibility and accept minimal duplicates. A possible mitigation in case that this becomes an issue would be to check the already scanned files for identical metadata (e.g. size, fragments) and re-scan them for the full data and compare it for equality. If an interface of the operating system is available, a lookup would be needed for each file if multiple hard links are present and checked, if the file has already been incorporated in the report.

2.2.2 Symbolic link

A symbolic link can link to any other path, including remote paths [29]. Both relative and absolute targets are allowed [29]. Avoiding them is simple for us because Java exposes an interface to check for this link.

2.2.3 Junction

A junction only exists in Windows and allows linking directories [28]. Links to other volumes on the same system are possible [28]. Other than that, a junction works identically like a hard link [28].

2.3 How to count fragments

Querying the API of operating systems yields either every allocated block one by one, or groupings of blocks called extents. Although extents represent a group of adjacent blocks, these extents cannot be taken as fragments, which was initially assumed. The extents have to be manually checked if they form fragments.

We start by examining the trivial case where a file has only one block which is illustrated by figure 2.1. This could be described as the most basic form of an extent.

Figure 2.1: An example file that consists of a single block A only. This is the trivial case where the extent equals the block.

Note that the letters on the blocks in the figures denote the ordering of the data. Data is written alphabetically and to reconstruct it has to be read in the same order. This rule is not needed for the first examples because all blocks are ordered, but we will take a look at this fact later.

Next, we examine the base-case, a file with two blocks where one block follows the previous one directly. Figure 2.2 illustrates this. Both blocks form a single extent.

Figure 2.2: An example file that consists of two blocks A and B. B follows A directly without any gap. Both form an extent.

Likewise, if there is a gap in-between, each block forms its own extent. No combined extent is established as illustrated in figure 2.3.

Figure 2.3: The same file as in figure 2.2 but with a gap between blocks A and B. Although B follows A it is not directly adjacent and therefore they form two separate extents.

This rule can be generalized to not only apply to individual blocks but also extents. One of the two blocks or even both can be replaced by extents and they would form a single compound extent.

Using this generalization viewing multiple extents or block as single unit, it becomes possible to group multiple blocks as one extent, as illustrated in figure 2.4.

Figure 2.4: The same check as in figure 2.2 is performed, except that A consists of two extents (or blocks) and is viewed as single unit. Therefore, extent A and B form a bigger combined extent.

Until now we only looked at blocks and extents but never touched fragments. A fragment is an extent that cannot be extended by taking all other remaining blocks or extents into account. In other words, an extent is a fragment if no other extent (or block) exists in the file that fulfills the rule that merges two extents together.

This would mean that a fragment can only consist of one extent. However, the result of a query on an operating system can yield extents that still can be combined further. This is what we described in the beginning of this section that manual checks are necessary.

There are no more rules or terms than the few we have defined now. We can take a look at a more complicated example consisting of four blocks. Figure 2.5 provides an overview over the structure of the example-file.

We start our analysis by recognizing that A and B are directly adjacent and thus form an extent. Using the generalization we take this single extent which precedes the directly adjacent block (or extent) C. These three blocks (or extents) now form a fragment because extent D follows C but is not directly adjacent and therefore not a part of it. D forms its own fragment, because there is no following directly adjacent block/extent.

Figure 2.5: A file consisting of 4 extents. The first three extents form the first fragment and the last extent forms a second one.

These rules provide the foundation of analyzing blocks, grouping them to extents and finding fragments. APIs return the extents in the same order as the data was written (sequentially), not the order on disk. In other words, the extents of a file might be in reverse order on disk than needed when reading the file from beginning to end sequentially.

Figure 2.6 shows the possibility of such a non-linear order on disk. The extent label letters represent the (lexicographical) order in which the data of the extents has to be read to reconstruct the originally saved data. The order from left to right represents the position on the disk[4].

Figure 2.6: A file with two fragments, (A,B) and (C). Although the data in C is at the end of the file, the data comes first on disk.

A trickier example is depicted by figure 2.7. This time, the last block (as seen from the contents of the file) is preceding and directly adjacent to the first block of the file. Both do not form an extent because the order is reversed and they are not alphabetically adjacent.

Figure 2.7: A file with two fragments, (A,B) and (C). Although the data in C is at the end of the file, the data comes first on disk. Since C precedes A (instead of follows) and also is not alphabetically adjacent no extent is formed.

In our last example, we return to a file with only two blocks as shown in figure 2.8. Although the two blocks are alphabetically adjacent to each other, again no extent can be formed because of the reverse order.

[4]We assume that the storage medium is flattened such that every data-point can be accessed by a single coordinate (LBA).

Figure 2.8: A file with two blocks A and B. Although directly adjacent on disk and logically by the stored data, no extent is formed because B precedes A instead of following it.

All examples above assume that every bit of data of a file is stored on disk. However, there are file systems that allow allocation of a file without any actual write operation. Thus, no actual blocks are reserved because there is no data stored in them.

As an example, a file could have data at the beginning and the end with a hole in the middle. Though, the file system can choose to place both parts adjacent to each other as depicted by figure 2.9.

Figure 2.9: A file with two blocks A and C. The block B is missing because there was no data written to it and therefore never allocated. Although both block are directly adjacent, no extent is formed because A and B are not alphabetically adjacent.

The main question is if we count the missing block B. From the perspective of the virtual file it is there but not allocated. On the other hand, from the point of the actually written blocks, the blocks neighboring the hole are adjacent and therefore do form an extent.

For our implementation we used the second approach where the gap is ignored and therefore a single fragment is formed. In this regard we followed the interpretation of *MyDefrag* helper tool which can be found in section 5.1.

3
Implementation

The project generally consists of two parts: The first part is the native implementation that contains C code that interfaces directly with the operating system. The second one contains the remaining parts ranging from using the native code gathering the fragmentation information up to the graphical interface and the uploader for fragmentation reports.

For easier handling, the whole project is implemented in layers where each layer uses the one directly below. At the bottom are the C implementations for the different operating systems. Figure 3.1 provides an overview over this layer-structure.

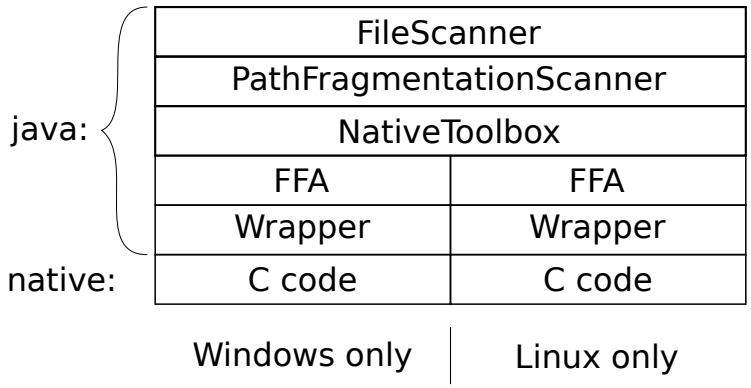


Figure 3.1: The structure of the implementation. Only the lowest layer is implemented in C, all other in Java. The implementations of the FileFragmentationAnalyzer (FFA) and below are operating system specific.

3.1 Native side

Normally, Java is a very restricted language, only allowing access outside the JVM through well-defined operating system independent interfaces. In order to be able to use native functions JNI (Java Native Interface) has to be used [30].

3.1.1 JNI in general

It works by defining methods as `native` and providing no body as if it was an abstract method [30]. An example for the Java definition is taken from the implementation where `CreateFile` of the winapi is called:

```
package me.nithanim.fragmentationstatistics.natives.windows;

public class WinapiNative implements Winapi {
    public native long createFile(String p);
    .
    .
    .
}
```

javah is a tool accompanying the java compiler *javac*. It takes classes that have such native methods as input and generates C header files from it [30].

The tool is only included in JDKs up to JDK9. Its functionality was merged into *javac* starting with JDK10 and the separate tool removed completely [31]. This complicates the building process for higher versions because a special Maven-plugin is used to compile the C code which is incompatible with this change.

The function names defined in this header are crafted in such a way that they are unique and one native java method has exactly one matching function on the native side [30]. A function name includes the package, class and method name [30].

The corresponding example for the example above would be the following function:

```
JNIEXPORT jlong JNICALL
    Java_me_nithanim_fragmentationstatistics_natives_windows_WinapiNative_createFile(JNIEnv
    *env, jobject obj, jstring path)
```

Additional java definitions were created for the necessary structs and also for helper classes.

Structs

For structs, simple wrapper classes were created that have accessor methods for every member. Additionally, methods for the management of the native allocation and especially the de-allocation (for which `AutoCloseable` is used) were implemented. The accessor methods are declared as `native`.

The pointer to the native struct is stored in a field in the java object. When an accessor is called, the address is retrieved, cast back to a pointer and then the requested operation performed.

An example for a struct would be the wrapper of `STARTING_VCN_INPUT_BUFFER`. The following listing contains the structures as defined in the header files:

```
typedef struct {
    LARGE_INTEGER StartingVcn;
} STARTING_VCN_INPUT_BUFFER, *PSTARTING_VCN_INPUT_BUFFER;
```

The manually created wrapper looks like:

```
public class StartingVcnInputBuffer extends AutoCloseable {
    public native long getStartingVcn();
    public native void setStartingVcn(long startingVcn);
    public native void close();
}
```

As an example, the getter for the starting VCN looks like this:

```
JNIEXPORT jlong JNICALL
    Java_me_nithanim_fragmentationstatistics_natives_windows_
    StartingVcnInputBufferNative_getStartingVcn(JNIEnv *env, jobject obj)
{
    jclass class = (*env)->GetObjectClass(env, obj);
    jfieldID fieldId = (*env)->GetFieldID(env, class, "addr", "J");
    STARTING_VCN_INPUT_BUFFER *b =  (STARTING_VCN_INPUT_BUFFER *)(void *)(*env)
        ->GetLongField(env, obj, fieldId);
    return (jint)b->StartingVcn.QuadPart;
}
```

Functions

Besides the simple structs, the native functions also need to be made accessible. For simplicity and because not a lot of methods are needed, a class is employed that bundles all native methods.

The access to the functions getting information about file fragmentation differs completely between Windows and Linux. MacOS might expose similar or even exactly the same interface but this was not explored for this thesis.

The general guideline while implementing was to keep as much in Java as possible. One advantage is the safety of a high-level language. Additionally, it is very cumbersome and error-prone to interact with Java from the C code.

Therefore, the only bigger C function we created allows us to conveniently throw Java exceptions from C. Since it only calls the JVM, it is shared across all operating systems.

```
throwNativeCallException(
    JNIEnv *env,
    const char *function,
    jlong returnCode,
    jlong errorCode)
```

If an error is encountered in a C function, it tells us the function name that we wanted to call, its result value and also the error code. One example using this helper would be `closeHandle`. No result is expected but we check the result and throw an exception in case of an error:

```
JNIEXPORT void JNICALL
    Java_me_nithanim_fragmentationstatistics_natives_windows_WinapiNative_
    closeHandle(JNIEnv *env, jobject obj, jlong handle)
{
    BOOL r = CloseHandle((HANDLE)handle);
    if (r == 0)
    {
        throwNativeCallException(env, "CloseHandle", r, GetLastError());
    }
}
```

This gives us valuable information in case of an error instead of a silent failure or even a crash.

Following the guideline of the least amount of C code, it is therefore often the case that the methods are not directly forwarded with JNI. Instead, a new `private native` method is created which takes care of calling the appropriate OS function. We often use this approach to have an advanced API on the Java side while keeping the native parameters and therefore the C code simple. Passing complex objects through JNI would require making Java method calls in C to convert the parameters to usable values. Therefore we carry them out on the Java side and only pass primitives and strings.

3.1.2 General interface

Although most of the code is different for the operating systems, some functionality was extracted into the common interface *FileSystemUtil*.

```
public interface FileSystemUtil {
    FileSystemInformation getFileSystemInformation(Path p);
    OperatingSytem getOperatingSystem();

    @Value
    public static class FileSystemInformation {
        String name;
        long magic;
        long totalSize;
        long freeSize;
        long blockSize;
    }

    public static enum OperatingSytem {
        LINUX,
        MACOS,
        WINDOWS
    }
}
```

It enables requesting information about the file system by the scanner without having to know which implementation to choose. This interface is implemented by the OS specific classes. At runtime the right implementation is selected once before the scan.

3.1.3 Windows

At first we are going into the details of the Windows side. We created an interface that outlines the methods we need to interact with Windows to extract all the information we are interested in:

```
public interface Winapi extends FileSystemUtil {
    long createFile(Path p) throws IOException;
    void closeHandle(long h);
    boolean fetchData(long fileHandle, StartingVcnInputBuffer inputBuffer,
        RetrievalPointersBuffer outputBuffer);
    StartingVcnInputBuffer allocateStartingVcnInputBuffer();
    RetrievalPointersBuffer allocateRetrievalPointersBuffer(int nElements);
}
```

The real implementation is done in *WinapiNative*. The first two methods, `createFile` and `closeHandle` are very simple and are directly forwarded with JNI. For the parameter values used, please refer to section 2.1.2.

`fetchData` fills the given output buffer with the given input buffer utilizing the handle to a file. Both buffer classes are only thin wrappers of their native counterparts as stated previously. It calls the `native` sub method `fillRetrievalPointers` to take care of the `DeviceIoControl` syscall:

```
@Override
public boolean fetchData(long fileHandle, StartingVcnInputBuffer inputBuffer,
    RetrievalPointersBuffer outputBuffer) {
    int r = fillRetrievalPointers(fileHandle, inputBuffer.getAddr(),
        outputBuffer.getAddr(), outputBuffer.getNumberAllocatedExtents());
    if (r != 0) {
        ... error checking
    } else {
        return true;
    }
}

private native int fillRetrievalPointers(long handle, long inputBufferAddr,
    long outputBufferAddr, int nAllocatedExtents);
```

After returning, the result is checked if the end of the file was reached or if the buffer was too small and another query has to be made to fetch the remaining extents. This is communicated by returning a boolean that tells the analyzer if all data was fetched (true) or if more is available (false);

The last two methods in the *Winapi* interface, `allocateStartingVcnInputBuffer` and `allocateRetrievalPointersBuffer`, simply return the appropriate wrapper object.

`getOperatingSystem`, inherited from FileSystemUtil, simply returns `WINDOWS`.

`getFileSystemInformation` is the longest and most complex function implemented in C. The major issue being that a lot of information is needed that is scattered across different functions.

Unlike in Linux where multiple structs and functions are used, Windows only uses pointers where the result is stored which do not exist in Java. Because of this, this is the only time in the project, where a Java data object (not counting the exceptions here) is created and passed back to the Java code.

`GetVolumePathName`, `GetVolumeInformation` and `GetDiskFreeSpace` are called and the file system name, total size, free space and block size passed to the constructor of the `FileSystemInformation` class. The newly created object is the returned from the C function. For information about the called Windows functions please refer to section 2.1.2.

3.1.4 Linux

For Linux we also defined an interface for all native methods and let it extend the `FileSystemUtil`. It is implemented by `LinuxApiNative`. The interface itself is a lot more complicated than the Windows one but the C code is a lot less complex. For information about the functions and parameters of the operating system please refer to section 2.1.3.

```
public interface LinuxApi extends FileSystemUtil {
    int openFileForReading(Path path) throws IOException;
    void closeFile(int fd);
    int getBlocksize(int fd);
    long getFilesystemType(Path path);
    void fstat(int fd, StatStruct stat);
    void stat(Path file, StatStruct stat);
    void statvfs(Path file, StatVfsStruct stat);
    FiemapStruct allocateFiemapStruct(int maxExtents);
    StatStruct allocateStatStruct();
    StatVfsStruct allocateStatVfsStruct();
    int fibmap(int fd, int idx);
    void fillFiemap(int fd, FiemapStruct fs);
}
```

`openFileForReading` and `closeFile` are directly passed to JNI and call the `open` and `close` syscalls respectively. `getBlocksize` is implemented calling `ioctl` with the `FIGETBSZ` control code.

`stat`, `fstat` and `statvfs` are directly calling the C function with the same name. The three allocation methods, `allocateFiemapStruct`, `allocateStatStruct` and `allocateStatVfsStruct`, are simply forwarding the invocation to the respective struct class that then allocates the native memory.

`getFilesystemType` internally uses `stat` to fetch the magic of the file system.

The last two methods, `fibmap` and `fillFiemap`, are the ones that retrieve the fragmentation data. Because of their simplicity, they do not require much C code, except for the call itself and some error checking. On the Java-side, `fibmap` it is even directly passed through.

For the inherited `getOperatingSystem` we always return `LINUX`. Again, the `getFileSystemInformation` is the most complex one, but since we are dealing with structs (instead of multiple result pointer like on Windows), we can handle the whole logic in Java code. At first we call `statvfs` to fill the `StatVfsStruct` that contains nearly all the information we need. Only the file magic has to be fetched with `stat` which we fetch with `getFilesystemType`. We query the interesting information from the `StatVfsStruct` and create the `FileSystemInformation` from it.

3.2 Java side (low level)

3.2.1 FileFragmentationAnalyzer

The *FileFragmentationAnalyzer* is the heart of the application. It is the common interface for an analyzer gathering information about the fragmentation of a specific file.

To support different operating systems, implementations need to be specifically programmed. For this thesis complete implementations for Windows and Linux were created.

```
public interface FileFragmentationAnalyzer extends AutoCloseable {
    List<Fragment> analyze(Path p) throws IOException;
}
```

The interface itself requires only one method named *analyze* taking a path to a file. The result is a list of fragments.

Since an implementation might be dealing with native memory, *FileFragmentationAnalyzer* extends *AutoCloseable*. This introduces a *close* method that allows to free allocated memory if needed. This make it also compatible for use with the resource-try-catch construct of Java.

WindowsFileFragmentationAnalyzer

The *WindowsFileFragmentationAnalyzer* is one of the operating system specific implementations available targeting Windows. It does not directly call the windows function but uses *Winapi* as a wrapper.

For the operation, `StartingVcnInputBuffer`, `RetrievalPointersBuffer` and the `Winapi` are needed and given via the constructor. The first buffer is used to specify from which virtual file position the extents should be queried and the second one holds the resulting extents. Both are thin wrapper-classes with native functions for accessing the corresponding native struct.

The first action of the *analyze* method is querying the bytes per sector and sectors per cluster because all functions return the size in clusters. This is only done once because we expect to stay on the same hdd which is enforced by the class using this analyzer. Directly after, the Winapi function *CreateFile* is called, returning a handle for the file to be able to read its metadata.

The main part of the function is wrapped in a loop that ensures that all extents can be retrieved. Since the receiving struct is of a fixed size, a file might have more extents than the struct can accept. Therefore, the query is repeated sufficient times while setting the *StartingVcn* of *StartingVcnInputBuffer* after the last retrieved extent. For the first iteration, the *StartingVcn* is set to *0* to start from the beginning of the file. The end of the extent stream can either be determined by the error code of the native function or by looking for a specific flag set on an returned extent. The first variant was chosen for this implementation.

The returned extents are directly passed on to an `ExtentToFragmentCombiner` which handles the checking and combining of fragments. The functionality is pretty simple as it checks the the passed extents if it matches the expected LCN (the calculated end of the previous extent). On equality, the extent is merged with the previous and in case of a mismatch, a new extent is created.

At the end of the *analyze* method, the result of the `ExtentToFragmentCombiner` is returned and the handle to the file closed.

LinuxFileFragmentationAnalyzer

There are two methods of retrieving information about the file fragmentation on Linux. Therefore this is a wrapper implementation utilizing the *Fiemap* and *Fibmap* implementations. Both implement a new interface that allows passing the block size such that it can be queried once and re-used.

```
public interface LinuxSubFileFragmentationAnalyzer extends AutoCloseable {
    public List<Fragment> analyze(File[1] f, int blockSize) throws Exception;
}
```

The `analyze` method is implemented such that the *Fiemap* implementation with `LinuxFiemapFileFragmentationAnalyzer` is tried first. In case of an error, a fall-back to `LinuxFibmapFileFragmentationAnalyzer` is used.

LinuxFiemapFileFragmentationAnalyzer

In the constructor, two `FiemapStruct`s are allocated - one for querying the number of extents and one for the extent data itself. Since the mode of operation is determined

[1]The `File` here is just a helper for convenience allowing to utilize the resource-try-catch construct of Java.

by the size field in the struct, it would be possible to use the same struct for both. However, this would require a changing array length variable which would introduce state management. We determined it is not only safer, but also more obvious in the Java code to call the function with the appropriate struct.

In the `analyze` method we start by calling `fillFiemap` with the file descriptor and the position struct (the one with the zero-length-array) to get the total number of extents. We take this as base-line on how many extents we expect. Also we set the start-position on where we want the extents in the data struct to zero.

We iterate in a loop until all extents have been retrieved. On every iteration we call `fillFiemap` with the data struct which gets filled at most to the array length. With `getMappedExtents` on the struct we get the number of returned extents and continue with iterating and extracting them one by one by invoking `getExtent` on the struct. Calling `getLogical`, `getPhysical` and `getLength` on the extent, we have all the information we need and pass that to a `ExtentToFragmentCombiner` that merges adjacent extents. After all extents of the struct have processed, the *start* field is updated in the position struct to the position after the last extent returned.

After all extents of the file have been retrieved, the calculated fragments are returned.

LinuxFibmapFileFragmentationAnalyzer

Because of the simplistic nature of *Fibmap*, the analyzer is also very simple. At first the number of blocks is calculated by querying the size with *stat* and then dividing the result by the block size.

In a loop starting from zero to the number of blocks (excl.) a call to `fibmap` with the block index parameter set to the iteration counter is made. The result is the physical position of the block which is then passed to a *BlockMerger* which directly calculates the fragments from it.

At the end, the result from the *BlockMerger* is returned.

3.3 Java side (high level)

In the last section, we defined the interfaces and implementations of analyzers for the different operating systems and approaches. In this section we introduce the classes that combine and make use of the implemented functionality.

3.3.1 NativeToolbox

First off, we define a class that allows us to create the appropriate `FileSystemUtil` and `FileFragmentationAnalyzer` to interface with the current operating system.

For this, the `NativeToolbox` was created that holds the objects for both interfaces. An instance is constructed by calling the static `create` method, that checks the current system and returns the appropriate objects. The checks themselves are delegated to `Platform` which determines the operating system based on the `"os.name"` system property of the JVM.

3.3.2 PathFragmentationScanner

One class that makes use of the `NativeToolbox` is the *PathFragmentationScanner*. It is used by the GUI to scan an arbitrary path for all files and their fragments. Since it is already very high in the hierarchy, it already contains a lot of code for communication with the GUI to report the status while scanning.

The heart of the *PathFragmentationScanner* is the `scan` method. At first, the `NativeToolbox` is created which powers the hole scanning process. We utilize the `Files.walkFileTree(Path,FileVisitor)` provided by Java standard library to walk down the path given by the user. The file visitor is implemented in the `FileScanner` to which the `NativeToolbox` is passed.

3.3.3 FileScanner

The *FileScanner* extends the provided `SimpleFileVisitor` from the standard library.

We override `preVisitDirectory` which we use to only scan the HDD that was selected by providing the root path. Since this is a standard Java API, it only provides a limited set of data to work with.

What we are able to check is if it is a *file*, *folder*, *symbolic link* or *"other"*. Therefore, we skip the complete subtree if a *symbolic link* or *"other"* is encountered.

Linux handling is a bit tricky here it is a normal procedure to mount another file system in a folder. However, a mountpoint does not fall in either of those two categories. A check with the `stat` syscall is implemented that checks the if the device is the same as the one of the given root path.

The other important method of the visitor is `visitFile`. It calls the analyzer and then pushes the result to the storage.

3.4 Storage file format

One functionality the program includes is saving and loading scan results. It is not only handy for the user who analyzed their disk. Since we want to pack and transmit the gathered information to the server, the same format can be used.

Especially the transmission over the network requires a data format that is able to pack the information as much as possible to save on network traffic. Exporting to multiple formats should also be possible to allow other programs to parse the data easily, but often at the cost of some storage space.

We implemented writers for some well-know formats such as CSV, and JSON. It is also possible compress the file using gzip. For minimal storage (and transmission) requirements, a custom binary format was created.

We did a quick evaluation of the space requirements of each format on a 256 GB SSD Ubuntu installation. *.ffi* is the extension of the custom format and *.gz* shows that the file is gzip compressed. The following table shows the results:

`.ffi.gz`	1.6 MB
`.csv.gz`	1.8 MB
`.json.gz`	2.5 MB
`.ffi`	3,9 MB
`.csv`	12.9 MB
`.json`	81.7 MB

Keep in mind that this data is only from one disk and one particular system. It is in no way representative and most likely differs widely between similar systems.

3.4.1 Stored data

Regardless of the chosen file format the stored data is equivalent in every case. The data is heavily aggregated to help anonymization.

Besides the fragment information itself, some metadata is stored. This includes the *operating system* under which the scan was executed and the *file system type* the data is from.

For the file system type, a name and/or magic are stored. Windows only operates with a name and therefore only the name is used while the magic is not and defaults to zero. On Linux, the magic is the same as returned by the system. The name is the manually looked-up magic on a (pre-defined) table but it might not be complete or useful. Because of our limited sample size we we did not commit to the name alone and therefore store both name (if found) and magic.

Additionally, the block size, total size and free space of the file system is saved.

The heart of the storage file is the data about the file fragmentation itself. Every scanned file has one entry with the following information:

1. Virtual file size
2. Number of fragments
3. Number of backtracks
4. Sparse file flag
5. Time of creation
6. Time of last modification
7. Time of last access

The *virtual file size* is the size of the data that is stored in it. For anonymization purposes, not the exact size is stored but rather rounded down to a value based on the original size. To being able to capture small but also big files, an exponential classification separation process was chosen with 2^x MiB where we set $x = 12$ as maximum for now. This means that the files are separated in a buckets. The first contains all files under 1 MiB, the second all between 1 and 2 MiB, the third all between 2 and 4, ... All files beyond 4096 MiB are in the same bucket. The classification should be tweaked depending on the use-case.

Instead of the exact details of all fragments, only the *number of fragments* is saved. *Number of backtracks* states how often a following fragment actually came before the current fragment. In other words, how often the starting block number of the following block was smaller than the current one. The *sparse file flag* states if the file had "holes" in it. Normally, it would be possible to simply divide the file size by the block size to get the number of allocated blocks. If the flag is set this could return wrong results, since some blocks are not physically allocated.

To help anonymization, the three times are not timestamps but rather offsets from the scan time. They are calculated using the difference of the timestamp in question and the timestamp of the start of the scan. Since this would still have incredible precision, the difference is converted to weeks (rounded down).

3.4.2 The custom binary format

The byte order is big endian.

The first data is the file magic consisting of the letters `FRAG` or (0x46524147). The magic is directly followed by the file format version which is a varint. This is a safeguard in case the format has to be changed for whatever reason in the future.

Next comes the id of the operating system where Linux is 0, MacOS is 1 and Windows is 2 stored in a single byte. Although, MacOS is currently not supported this might change and was therefore included.

Directly following are the file system magic and name where the magic is a fixed unit64 and the name is the byte-length as short followed by *length* bytes. If unused, the string is zero-length and the magic is zero. The block size is given in bytes and stored as varint. The total size and free space are given in number of blocks (to save space) and are also varints.

Finally, the interesting payload starts containing the gathered file data. A varint holds the number of entries to expect. All the file-data stated previously (file size, fragments, ...) are saved as varints. The file size is given in blocks.

Varint

A varint (variable length integer) is a way to encode an integer (of arbitrary size) in as little bytes as possible [32]. This works by using the most significant bit of a byte as flag to tell if the following byte is part of the integer and contains more data [32]. Having the 8th bit as flag leaves the remaining 7 bits as actual payload for a single byte [32].

For our application this reduction plays a crucial role. As an example, the (rounded) file size (in bytes) is transmitted, which can range from 0 bytes all the way up to one terabyte or more. Using fixed size integer this would require transmitting 8 bytes in every case, even when the file size is only 650 bytes. Using varints the transmission would only require 2 bytes.

We follow up by providing a full encoding example based on the 650 bytes long file. We look at this number n in binary representation:

```
n:  0000 0010   1000 1010
```

We take the lower 7 bits as our next value to write to the stream which we name v. We remove the bits from the original number and move the remaining bits 7 to the right resulting in:

```
n:  0000 0000   0000 0101
v:               000 1010
```

We then check if there is still data to transmit (bits set to 1) which it is in this example (`101` is left). Therefore, we set the most significant bit (MSB, the 8th bit) of v to 1 and send it off to the stream indicating for the reader that more data is coming:

```
1000 1010
```

Next, we start again taking the lower 7 bits, saving them in v and shifting the remaining right by 7.

```
n:  0000 0000   0000 0000
v:              0000 0101
```

This time there are no bits to send left in n so we send off v to the stream directly and terminate the loop.

Reading a varint works the same way but backwards. After a byte is read, the MSB is separated from the 7 data bits. The latter is appended to the data already received (as in shifted to the higher bits based on how much data was already received). If the MSB is is set, another iteration with the next byte is made. Otherwise, the the integer is reconstructed.

4
Tests

Building test-cases for the program was not as straight forward as it might normally be. Working with files, one would simply create a test-file with appropriate testing code. The algorithm would be applied to the file and then the test would check the output of the tested code against the expected output.

In our case creating a file and and querying the fragments would yield different values on different machines/files systems because the cluster indices would be different. Theoretically, the observed values could change over time without accessing the file as the file system itself or another program (e.g. defragmentation program) might move blocks around.

A possible solution would be to access the same API as a defragmentation program and move the blocks of a test file in the same position that they were in when the test case was created. This would require understanding of the API and full implementation of such a process. Sparse files would require additional handling. This would also need an implementation for every for every operating system.

Because implementing such a specifically fragmenting system would result in a complex system that would require tests in itself, we settled for a simpler approach. This however means, that we effectively skip testing the absolute low-level (native) functions that access the the operating system. However, there is a way to test these functions by comparing the output of a native debug utility built for the running operating system.

4.1 High-level functions

We refer to all functions that work with the output of the native functions as *high-level functions*. We will base the explanation of the testing-process on the Windows implementation but it is equally applicable for Linux.

Since all native functions are combined in a single class, originally `Winapi`, the first step was to extract an interface from it. The interface took over the original name and the native implementation was moved to `WinapiNative`. This allows another implementation of `Winapi` specifically for testing which is named `WinapiTesthelper`.

The `WinapiTesthelper` imitates the the native functions. The high-level code that normally calls the native functions with `WinapiNative` is then redirected to `WinapiTesthelper`. Based on the function called and the given parameters, the result matches the one that would be returned by the operating system function directly.

The values that are returned are read from a file. It is created manually by directly saving the results of the native functions. The file contents can be transferred and fetched independently from the current operating system, making the output deterministic. While this allows testing the high-level code on Windows, it also allows testing it on other operating systems because no native functions are called.

This allows us to test the the `FileFragmentationAnalyzer`. The following listing shows a part of the testing code for Windows:

```
WinapiTesthelper winapi = winapiFromFile(p, file);
StartingVcnInputBufferTesthelper inputBuffer =
    new StartingVcnInputBufferTesthelper();
RetrievalPointersBufferTesthelper outputBuffer =
    new RetrievalPointersBufferTesthelper(10);
WindowsFileFragmentationAnalyzer analyzer =
    new WindowsFileFragmentationAnalyzer(winapi, inputBuffer, outputBuffer);

List<Fragment> actual = analyzer.analyze(p);
List<Fragment> expected = JavaTestHelper.fragmentsFromResourceWindows(file);
Assert.assertEquals(expected.size(), actual.size());
for (int i = 0; i < actual.size(); i++) {
    Assert.assertEquals(expected.get(i), actual.get(i));
}
```

At first the *Winapi* is initialized which uses the data from a file as data source instead of querying the OS. The both buffers are fully implemented in Java which is easy since the C structs only hold values.

The analyzer is created with the three objects and the called to analyze the file. The expected values are also loaded from a file and then compared against the results.

4.2 Native functions

This is a tricky subject as stated previously. From testing the high-level code we now assume the algorithm that converts the results of the native functions to fragments to be correct.

The approach is to call the high-level code using native functions on a real file. The result (the fragment data) is then compared with the output of an external program also analyzing the same file. A collection of these programs can be found in chapter 5.

Verifying equality of the results is done manually for now until a better solution is available. One solution would be moving the extents of the test file to the appropriate locations which is complicated, quite invasive and sometimes impossible (e.g. smaller disk or unmovable extents). Another possibility would be either parsing the output of the external program or imitating its output checking the equality exactly.

5
Tools

5.1 MyDefrag

MyDefrag is/was a scriptable defragmentation tool with various pre-shipped scripts for different modes of operation.

The package of an old version contained the valuable helper program named *MyFragmenter.exe* which could be used for debugging. It was capable of not only analyzing and displaying fragments of a specific file but also deliberatly fragmenting an existing file or creating a new one with specified size and number of fragments. We could not find this program in any current release.

We are not sure about the the timeline on the program name and websites but it seems that it had undergone changes multiple times. The websites seem to have changed multiple times and the one of the original author (http://www.kessels.com/)< is broken. The official page http://http://www.mydefrag.net/[1] does not contain a lot of information. What we did find was that old versions of the website were still accessible via https://web.archive.org/ querying http://www.mydefrag.com/. Even the download was archived but we did not test it. It might be worth a try if such a program is needed.

Running the program the output was very detailed and useful for debugging the implementation. The listed output here is cropped, because it prints the rather large help every time and also the output was very large for the given file:

```
MyFragmenter v1.2, 2008 J.C. Kessels
...
Processing: Roberts Space Industries\StarCitizen\LIVE\Data.p4k
Fragment list:
  Extent 1: Lcn=105822180, Vcn=0, NextVcn=10784
  Extent 2: Lcn=105832964, Vcn=10784, NextVcn=20501
  Extent 3: Lcn=105842681, Vcn=20501, NextVcn=30796
...
  Extent 134 (virtual): Vcn=3588800, NextVcn=9965488
  Extent 135: Lcn=130446545, Vcn=9965488, NextVcn=9970224
  Extent 136: Lcn=104165090, Vcn=9970224, NextVcn=9992912
```

[1] http://mydefrag.8qm.de/ and http://jkdefrag.8qm.de/ are linked from there.

```
  3531984 clusters, 18 fragments.

Finished, 1 files processed.
```

5.2 filefrag

Filefrag is a small utility tool to quickly check the fragmentation of a specific file [23]. Using the simplest mode of operation it just displays the give path of the file along with the number of extents.

The most helpful flag for us is `v` that enables highly verbose output in addition to the default printing [23]. The result shows the files system magic and a big table with a row for every extent. The columns include the logical and physical ranges (beginning and endings), the length and flags. Additionally, the expected physical offset is printed which shows where the extent must have started at such that it could have been merged with the previous extent.

The following listing shows a sample output of *filefrag* on a test-file using the verbose flag:

```
Filesystem type is: ef53
File size of testfile.bin is 7630144 (1863 blocks of 4096 bytes)
 ext:     logical_offset:        physical_offset: length:   expected: flags:
   0:        0..     511:   56753152..   56753663:    512:
   1:      512..     827:   56730112..   56730427:    316:   56753664:
   2:      828..    1862:   56755688..   56756722:   1035:   56730428: last,eof
testfile.bin: 3 extents found
```

In this output we can see that for every extent (after the first one) the expected physical offset was not the one that was expected. Therefore every extent in this example file forms a fragment on its own.

5.3 hdparm

Besides *filefrag* *hdparm* can also be used for querying the file fragments. Although, the program provides a wide range of functions to interact with SATA/IDE devices, there is also a *fibmap* parameter [33].

It returns similar results as *filefrag* but it is special in that case that the physical position is given from the start of the HDD itself instead of the file system [33]. Contrary to the name of the parameter, *Fiemap* is tried first and used *Fibmap* as fallback [33].

Based on the example output using the same file, we can see the similarities to *filefrag*:

```
testfile.bin:
 filesystem blocksize 4096, begins at LBA 0; assuming 512 byte sectors.
 byte_offset  begin_LBA    end_LBA    sectors
           0  454025216  454029311       4096
     2097152  453840896  453843423       2528
     3391488  454045504  454053783       8280
```

5.4 Comparison of the outputs

We have seen some sample outputs of these tools. We now want to compare them to our implementation and generate outputs of the same file with each of our approaches (Fiemap, Fibmap and Windows).

5.4.1 Linux

At first we compare the two tools to each other. We can verify the the equality of the outputs only to some extent because *hdparm* has different physical offsets but the lengths of the extents and the byte offsets (logical offset) must be the same. The output of both tools was:

```
block size: 4096 bytes
filefrag:
  ext:     logical_offset:        physical_offset: length:   expected: flags:
  0:        0..     511:   56753152..   56753663:    512:
  1:      512..     827:   56730112..   56730427:    316:   56753664:
  2:      828..    1862:   56755688..   56756722:   1035:   56730428: last,eof
hdparm:
  byte_offset  begin_LBA    end_LBA    sectors
            0  454025216  454029311       4096
      2097152  453840896  453843423       2528
      3391488  454045504  454053783       8280
```

Multiplying the logical offsets of *filefrag* by the 4096 byte block size yields 0, 2097152 and 3391488 bytes which matches the byte offsets of *hdparm*.

We do the same with the lengths which yields 2097152, 1294336 and 4239360 bytes for *filefrag*. Since *hdparm* lists the number of sectors, we also need to multiply the values by the assumed sector size, resulting in 2097152, 1294336 and 4239360. Both outputs match exactly.

When comparing the results to our implementation we need to keep in mind that the existing tools return extents and not fragments. Fortunately, we can tell from the output of *filefrag* that each extent does form a fragment because the *expected* column is filled on every extent. This enables us direct comparison to our program which reports fragments, not extents.

We start examining the output of our implementation by printing the list of fragments produced by the *LinuxFiemapFileFragmentationAnalyzer*:

```
Fragment(offset=0, diskOffset=232460910592, size=2097152)
Fragment(offset=2097152, diskOffset=232366538752, size=1294336)
Fragment(offset=3391488, diskOffset=232471298048, size=4239360)
```

We can see that the in file offset and the size of the extents match by re-using the values we already calculated when comparing the two tools. To compare the physical offsets, we multiply the starting physical offset of *filefrag* by the block size and get 232460910592, 232366538752 and 232471298048. This is the same as the `diskOffset` of our implementation.

Now we execute the *LinuxFibmapFileFragmentationAnalyzer* and get the following result:

```
Fragment(offset=0, diskOffset=232460910592, size=2097152)
Fragment(offset=2097152, diskOffset=232366538752, size=1294336)
Fragment(offset=3391488, diskOffset=232471298048, size=4239360)
```

Comparing this to the output of the *LinuxFiemapFileFragmentationAnalyzer*, we can see that both are equal.

5.4.2 Windows

To create a Windows example, we used the *MyFragmenter.exe* to fragment an existing file into three fragments. The output looks as follows:

```
Fragment list:
  Extent 1: Lcn=99344761, Vcn=0, NextVcn=19618
  Extent 2: Lcn=114416022, Vcn=19618, NextVcn=39236
  Extent 3: Lcn=72970054, Vcn=39236, NextVcn=58852
  58852 clusters, 3 fragments.
```

The cluster size is not listed but we can calculate it by looking at the number of clusters and the file size on disk of 241057792 bytes (via the properties window) resulting in a cluster size of 4096 bytes.

Running our implementation yields the following output:

```
Fragment(offset=0, diskOffset=406916141056, size=80355328)
Fragment(offset=80355328, diskOffset=468648026112, size=80355328)
Fragment(offset=160710656, diskOffset=298885341184, size=80347136)
```

To be able to compare the results we multiply the output of *MyFragmenter.exe* by the cluster size. We also subtract the nextVcn with the currentVcn to calculate the length and then multiply this too. This yields the following result:

```
Vcn=0, Lcn=406916141056, Length=80355328
Vcn=80355328, Lcn=468648026112, Length=80355328
Vcn=160710656, Lcn=298885341184, Length=80347136
```

Comparing the values of *MyFragmenter.exe* to the output of our implementation shows, that all values match.

6
Outlook

In the introduction we talked about how fragmented files hinder file recovery or digital forensics. Personally, having file recovery done for friends and family, most interesting files were images but also documents.

Thinking about images, most were personal pictures taken which were saved once and never modified afterwards. This might be beneficial for less fragmented files because they are only written once and if written fragmented they might be picked up by an automated defragmentation process at some point.

Documents on the other hand are hard to predict because I have seen sizes ranging from a few KiB with just a little bit of text up to many MiB with many pictures. Additionally, the files are either written to in a short period of time or read only. The write operations most likely lead to changes in the files size size. My general assumption is that they grow over time while more text is added. This would introduce more and more fragments if the space where the file would be extended to is already in use.

Therefore, it would be interesting to see, how bigger but static images differ to smaller, but potentially steadily growing files in terms of fragments.

7
Bibliography

[1] H. T. Sencar and N. Memon, "Identification and recovery of jpeg files with missing fragments," *digital investigation*, vol. 6, pp. S88–S98, 2009.

[2] W. Davy, "Method for eliminating file fragmentation and reducing average seek times in a magnetic disk media environment," 1998, uS Patent 5,808,821.

[3] N. Memon and A. Pal, "Detecting a file fragmentation point for reconstructing fragmented files using sequential hypothesis testing," 2013, uS Patent 8,407,192.

[4] R. Smith, "Linux on 4 kb sector disks: Practical advice," https://developer.ibm.com/tutorials/l-linux-on-4kb-sector-disks/, accessed: 2018-11-21.

[5] Seagate, "Transition to advanced format 4k sector hard drives," https://www.seagate.com/gb/en/tech-insights/advanced-format-4k-sector-hard-drives-master-ti/, accessed: 2019-01-25.

[6] "Ext4 disk layout," https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout, accessed: 2018-11-02.

[7] Microsoft, "Default cluster size for ntfs, fat, and exfat," https://support.microsoft.com/en-us/help/140365/default-cluster-size-for-ntfs-fat-and-exfat, accessed: 2018-06-16.

[8] Microsoft, "Sparse files," https://docs.microsoft.com/en-us/windows/desktop/fileio/sparse-files, accessed: 2018-11-09.

[9] Microsoft, "Sparse file operations," https://docs.microsoft.com/en-us/windows/desktop/fileio/sparse-file-operations, accessed: 2018-11-09.

[10] Microsoft, "Clusters and extents," https://msdn.microsoft.com/en-us/library/windows/desktop/aa363841(v=vs.85).aspx, accessed: 2018-06-16.

[11] Microsoft, "Volume management," https://msdn.microsoft.com/en-us/library/windows/desktop/aa365728(v=vs.85).aspx, accessed: 2018-06-16.

[12] Microsoft, "Retrieval_pointers_buffer structure," https://msdn.microsoft.com/en-us/library/windows/desktop/aa365521(v=vs.85).aspx, accessed: 2018-06-16.

[13] Microsoft, "Createfilea function," https://docs.microsoft.com/en-us/windows/desktop/api/fileapi/nf-fileapi-createfilea, accessed: 2018-12-13.

[14] Microsoft, "Closehandle function," https://msdn.microsoft.com/en-us/library/windows/desktop/ms724211(v=vs.85).aspx, accessed: 2018-12-13.

[15] Microsoft, "Fsctl_get_retrieval_pointers control code," https://docs.microsoft.com/en-us/windows/desktop/api/ioapiset/nf-ioapiset-deviceiocontrol, accessed: 2018-12-13.

[16] Microsoft, "Fsctl_get_retrieval_pointers control code," https://msdn.microsoft.com/en-us/library/windows/desktop/aa364572(v=vs.85).aspx, accessed: 2018-12-13.

[17] Microsoft, "Starting_vcn_input_buffer structure," https://msdn.microsoft.com/en-us/library/windows/desktop/aa365673(v=vs.85).aspx, accessed: 2018-11-09.

[18] Microsoft, "Retrieval_pointers_buffer structure," https://docs.microsoft.com/en-us/windows/desktop/api/winioctl/ns-winioctl-retrieval_pointers_buffer, accessed: 2018-12-16.

[19] Microsoft, "Getvolumepathnamew function," https://docs.microsoft.com/en-us/windows/desktop/api/fileapi/nf-fileapi-getvolumepathnamew, accessed: 2018-12-13.

[20] Microsoft, "Getvolumeinformationa function," https://docs.microsoft.com/en-us/windows/desktop/api/fileapi/nf-fileapi-getvolumeinformationa, accessed: 2018-12-13.

[21] Microsoft, "Getdiskfreespacea function," https://docs.microsoft.com/en-us/windows/desktop/api/fileapi/nf-fileapi-getdiskfreespacea, accessed: 2018-12-13.

[22] L. Torvalds, "Fiemap ioctl," https://www.kernel.org/doc/Documentation/filesystems/fiemap.txt, accessed: 2018-12-19.

[23] F. S. Foundation, "filefrag - report on file fragmentation at linux.org," http://man7.org/linux/man-pages/man8/filefrag.8.html, accessed: 2018-07-01.

[24] F. S. Foundation, "ext2/e2fsprogs.git: misc/filefrag.c," https://git.kernel.org/pub/scm/fs/ext2/e2fsprogs.git/tree/misc/filefrag.c, accessed: 2019-01-07.

[25] R. Gooch and P. Enberg, "Overview of the linux virtual file system," https://www.kernel.org/doc/Documentation/filesystems/vfs.txt, accessed: 2018-12-19.

[26] F. S. Foundation, "stat," http://man7.org/linux/man-pages/man2/fstat.2.html, accessed: 2018-12-19.

[27] F. S. Foundation, "statvfs," http://man7.org/linux/man-pages/man3/statvfs.3.html, accessed: 2018-12-19.

[28] Microsoft, "Hard links and junctions," https://docs.microsoft.com/en-us/windows/desktop/fileio/hard-links-and-junctions, accessed: 2018-12-30.

[29] Microsoft, "Creating symbolic links," https://docs.microsoft.com/en-us/windows/desktop/fileio/creating-symbolic-links, accessed: 2018-12-30.

[30] C. Ullenboom, *Java 7 – Mehr als eine Insel*. Rheinwerk, 2011.

[31] J. Gibbons, "Jep 313: Remove the native-header generation tool (javah)," https://openjdk.java.net/jeps/313, accessed: 2018-11-21.

[32] Google, "Base 128 varints," https://developers.google.com/protocol-buffers/docs/encoding#varints, accessed: 2018-11-21.

[33] F. S. Foundation, "hdparm - get/set sata/ide device parameters," http://man7.org/linux/man-pages/man8/hdparm.8.html, accessed: 2019-01-09.